

FEATURES

AUTOMATING ASSOCIATION IMPLEMENTATION IN C++

by David M. Papert

David compares several unidirectional and bidirectional pointer-based methods for implementing one-to-one associations: a direct, handwritten implementation, a modular approach exploiting inheritance, and template-based implementations.

OBJECT-ORIENTED FACILITIES IN ADA 95

by David L. Moore

With the adoption of the ISO/ANSI Ada 95 standard, Ada supports object-oriented programming features such as class-wide objects, private types and child packages, multiple inheritance, and more.

PARTIAL REVELATION AND MODULA-3

by Steve Freeman

When compared to strongly typed languages, Modula-3 gives you greater flexibility in class reuse. Steve examines Modula-3's type system and describes how you can take advantage of its power.

OBJECT-ORIENTED PROGRAMMING IN S

by Richard Calaway

S, a high-level, object-oriented language, was originally designed for data analysis and graphics. As Richard points out, however, the S language is useful for a wide range of applications.

COBOL '97: A STATUS REPORT

by Henry Saade and Ann Wallace

The proposed Cobol 97 standard includes object-oriented features such as class definition, subclassing, data encapsulation, and polymorphism. Our authors focus on the object-oriented extensions to Cobol, and cover other proposed features.

18

28

36

44

52



FILE-STREAMING CLASSES IN C++

by Kirit Saelensminde

Kint implements a C++ file-streaming system that, unlike MFC or OWL, doesn't require a common superclass. This approach leads to less overhead and greater portability.

INSIDE MFC SERIALIZATION

by Jim Beveridge

The Microsoft Foundation Class Library implements a typesafe serialization mechanism that is both fast and flexible. Jim takes an in-depth look at how this mechanism works—and examines how you can get the most out of it.

58

62

EMBEDDED SYSTEMS

INSIDE FLASH MEMORY

by Brian L. Dipert

Direct-execute flash memory systems don't require the gigabyte hard disks and 64-Mbit DRAM arrays common in desktop systems.

68

NETWORKED SYSTEMS

ENVIRONMENT VARIABLES AND WINDOWS 3.1

by John (Fritz) Lowrey

Nothing can give network administrators a headache like several hundred networked computers which need to run software that demands customized run-time environments. To make the job even more difficult, using a single environment space in the Windows system area for all programs is dangerous, not to mention that environment handling under Windows 3.1 is poorly documented.

84

EXAMINING ROOM

EXAMINING CA-VISUAL OBJECTS

by Rod da Silva

CA-Visual Objects is an application-development environment that sports an incremental, native-code compiler, visual painters and editors, an advanced, active, repository-based storage system, and an object-oriented language that allows for optional strong typing and full object orientation.

90

PROGRAMMER'S WORKBENCH

POWERBUILDER NVOs

by Mark Robinson

Quite possibly, Non-Visual User Objects (NVOs) are Powerbuilder's most useful tool for creating true object-oriented applications. Mark examines the effective use of NVOs and their role in application development.

103

COLUMNS

PROGRAMMING PARADIGMS

by Michael Suaine

Michael takes a look at Sun's HotJava Web browser, as well as Java, the programming language it was written in.

131

C PROGRAMMING

by Al Stevens

Al wraps up "Midifitz," a Windows-based C++ program that uses MIDI to emulate a jazz piano player's rhythm section in real time. Midifitz examines the notes being played, deduces a musical chord, and plays a bass line through the MIDI system.

137

ALGORITHM ALLEY

edited by Bruce Schneier

This month, Louis Plebani examines search procedures that enable you to obtain the common fraction approximation of real numbers. In doing so, he focuses on number theories such as Farey Sequences.

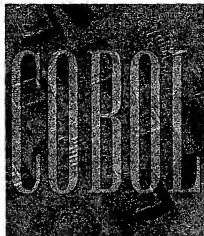
145

PROGRAMMER'S BOOKSHELF

by Ray Valdés

A good C++ book can be hard to find. Ray looks at some of the better ones.

149



FORUM

EDITORIAL

by Jonathan Erickson

6

LETTERS

by you

10

SWAINE'S FLAMES

by Michael Suaine

168

PROGRAMMER'S SERVICES

OF INTEREST

by Monica E. Berg

164

SOURCE CODE AVAILABILITY

As a service to our readers, all source code is available on a single disk and online. To order the disk, send \$14.95 (California residents add sales tax) to Dr. Dobb's Journal, 411 Boitel Ave., San Mateo, CA 94402, call 415-655-4100 x5701, or use your credit card to order by fax, 415-358-9749. Specify issue number and disk format. Code is also available through the DDJ Forum on CompuServe (type GO DDJ), via anonymous FTP from site ftp.mv.com (192.80.84.3) in the /pub/ddj/ directory, on the World Wide Web at <http://www.ddj.com>, and through DDJ Online, a free service accessible via direct dial at 415-358-8857 (1200/2400/9600 baud, 8-N-1).

NEXT MONTH

From the front cover to the back page, we'll be covering client/server architectures in our November issue.

Inside MFC Serialization

Typesafe serialization that's fast and flexible

Jim Beveridge

Having seen several commercial software packages through complete life cycles, I am reluctant to rely on "black-box" solutions, which tend to break down as a package evolves and becomes more complex. When I first saw the serialization mechanism in the Microsoft Foundation Classes (MFC), I questioned whether it was robust and flexible enough for a commercial application. I discovered that, although it has limitations, the serialization mechanism in MFC is strongly grounded in modern, object-oriented design theory. Furthermore, it is typesafe and leaves room for your design to evolve.

Using MFC serialization is straightforward. By default, any class derived from *CObject* can include a *Serialize()* member function that takes a *CArchive* as a parameter. In this member function, you add your own code to save and load any data associated with your class.

Data is serialized to and from a *CArchive* with *operator<<* and *operator>>*, much like *iostream*. The big difference is that *CArchive* is strictly a binary data format. As in *rostream*, there are default implementations that read and write fundamental data types such as *long* and *char*. The absence of the data type *int* facilitates portability between 16- and 32-bit implementations. The default implementations also handle byte swapping for types that support it. (For more information on portability between Little- and Big-endian architectures, see "Endian-Neutral Software,"

Jim, a software developer at Turning Point Software, can be contacted at jim@turningpoint.com.

by James R. Gillig, *DDJ*, October/November 1994.)

MFC's implementation seemed obvious and uninteresting until the day I created multiple document types in the same application. At that point, I noticed that whenever I loaded a file, MFC would correctly create the right kind of document object and call the proper *Serialize()* member function. This happened in spite of the fact that I had not written any code to help MFC create these document types. Or so I thought....



Problems, Problems Everywhere

To create a document or any other kind of object on the fly, MFC needs to solve three problems:

Problem 1. Arbitrary types must be created as needed, but the *new* operator can only create an explicit type, so a form of "virtual constructors" for *CObject* is necessary.

Problem 2. Developers need to be able to easily add new classes to be created. Ideally, this would be done in the class definition and/or implementation.

Problem 3. A mapping scheme is needed to allow a particular type to be created based on information read from a file. This mapping cannot be hardcoded in

MFC because developers add new types all the time.

As you'll see, MFC solves these problems elegantly with the use of a registry with automatic type registration and an implementation of virtual constructors based on these registered types. MFC's run-time type information is a building block for this architecture.

The Type Registry

To handle run-time type information, MFC creates a registry of classes in the application that are derived from *CObject*. This has nothing to do with the OLE registry, but the concept is similar. The type registry is a linked list of *CRuntimeClass* structures, in which each entry describes a *CObject*-derived class in the application. Listing One shows the *CRuntimeClass* structure (listings begin on page 122).

The real magic is that the types in this registry are not hardcoded in any table. The first clue to how this trick is accomplished is at the top of *SCRIBDOCH* from the MFC "Scribble" sample application. The beginning of the class declaration looks like Example 1(a).

The online help says to use the *DECLARE_DYNCREATE* macro to enable objects of *CObject*-derived classes to be created dynamically at run time. Although this is a concise description of what *DECLARE_DYNCREATE* does, what really goes on inside the macro is far more interesting. After preprocessing, the *DECLARE_DYNCREATE* macro expands to several new class members; see Example 1(b). (Note that all examples are reformatted all preprocessor-generated code for readability.)

The *GetRuntimeClass()* virtual function is the basis for run-time types in MFC. Run-time type information can be accessed for any *CObject*-derived object that includes *DECLARE_DYNAMIC*, *DECLARE_DYNCREATE*, or *DECLARE_SERIAL*. This information lets you determine if an object can legally be "downcast" to a derived

(continued from page 62)

class or if one object is the same class as another. Although Visual C++ does not support the new C++ operator *dynamic_cast*, using this run-time type information will achieve the same effect.

The run-time type information is declared using a static member variable, in this case *classCscribDoc*. The name (which has no space) is created in the various *DECLARE_xxx* macros with the macro-concatenation operator. Both *_GetBaseClass()* and *GetRuntimeClass()* are used to access this run-time class information. *GetRuntimeClass()* is virtual, so the type of an object can be found even with a pointer to a generic *CObject*.

Finally, the *Construct()* static member function forms the basis of MFC's use of its class registry as a class factory that can create arbitrary types on demand. To understand the workings of *Construct()*, some background is necessary.

Creating an Object

In *Advanced C++: Programming Styles and Idioms* (Addison-Wesley, 1992), James O. Coplien describes the concept of a virtual constructor:

The virtual constructor is used when the type of an object needs to be determined from the context in which the object is constructed.

In MFC, the context is based on information read from a serialized archive. However, a virtual constructor is only a concept, no language construct implements it directly. The *new* operator requires an explicit class as its argument. Virtual constructors can be created by implementing in each class a static function that calls *new*. This static member function can be called when a particular type is needed.

In MFC, this member function is called *Construct()*. It is created by the *IMPLEMENT_DYNCREATE* or *IMPLEMENT_SERIAL* macros. One of these macros must appear exactly once in a .cpp module for each class supporting dynamic creation.

In Scribble, the *IMPLEMENT_DYNCREATE* (*CscribDoc*, *CDocument*) statement appears near the top of *SCRIBDOC.CPP*. The first argument is the class, and the second is the class's parent class. Listing Two is the code generated by the preprocessor.

When MFC needs a document or any other *CObject*-derived type, it calls the *CreateObject()* member function for an instance of a *CRuntimeClass*. *CreateObject()* allocates the memory using the size in the

for this object. Again, the memory was already allocated by *CreateObject* using the size information in *CRuntimeClass*.

By using the registry of *CRuntimeClasses* and the *Construct()* member function, MFC is able to lookup and create new types on the fly, which solves Problem 1. A potentially serious problem with this technique is that multiple inheritance and virtual base classes are not supported (see MFC Technical Note #16).

Type Registration

Problem 2 is that users must be able to easily add new classes into the registry. The idea of types registering themselves is core to object-oriented design. If a type registers its own existence with a registry instead of hardcoding the type into the registry, then the type can be freely added and removed from the program without any code changes to the registry.

Although not immediately obvious, it is the *IMPLEMENT_DYNCREATE* macro that enables users to easily add new classes into the registry. In the expansion of *IMPLEMENT_DYNCREATE* in Listing Two, the static instance of *CRuntimeClass* in *CscribDoc* is initialized as in Example 3.

Several of these entries have already been discussed. In particular, the statement *sizeof(CscribDoc)* is used by *CreateObject* to allocate memory; then the function pointed at by *CscribDoc::Construct* initializes the memory.

The next line places this information into the MFC type registry: *static const AFX_CLASSINIT _init_CscribDoc(&CscribDoc::classCscribDoc)*; this declaration constructs an object of type *AFX_CLASSINIT* using a constructor that takes a *CRuntimeClass* as its argument. Because this object is at file scope, it will be constructed before *main()*. *AFX_CLASSINIT*'s constructor links this *CRuntimeClass* into the MFC type registry. *AFX_CLASSINIT* itself has no member data, so it will not take up any data space.

```
void _stdcall CscribDoc::Construct
(void* p)
{
    new(p) CscribDoc;
}
```

Example 2: The *Construct()* function.

```
CRuntimeClass CscribDoc::
classCscribDoc = {
    "CscribDoc",
    sizeof(CscribDoc),
    0xFF,
    CscribDoc::Construct,
    &CscribDoc::GetBaseClass,
    0};
```

Example 3: Initializing the static instance of *CRuntimeClass* in *CscribDoc*.

The type registry is a linked list of CRuntimeClass structures

CRuntimeClass structure, then calls *ConstructObject()*. *ConstructObject* verifies that the type supports dynamic construction, then calls the *Construct()* function.

Although no explanation is given in the sources, this arrangement cleanly separates the construction of an object from the memory allocation. This seems like a lot of extra work, but it is required under certain circumstances. For example, if an array were created manually, the memory would have to be created with a single *malloc()* in order to be contiguous. By using *ConstructObject()*, you could manually initialize each array entry. This mechanism allows decisions normally made in C++ at compile time to be made at run time.

Example 2 shows the *Construct()* function. The syntax of the call to *new* is a little unusual. The function actually called is *CObject::operator new(size_t, void*)*. Remember, the size of a structure is an implied argument when operator *new* is called, but must be explicitly declared in the operator *new* definition. This version of *new* in *CObject* does nothing, but calling *new* has the side effect of calling the construc-

```
(a) class CscribDoc : public CDocument
{
protected: // create from serialization only
    CscribDoc();
    DECLARE_DYNCREATE(CscribDoc)
};

(b) protected:
    static CRuntimeClass* _stdcall _GetBaseClass();
public:
    static CRuntimeClass classCscribDoc;

    virtual CRuntimeClass* GetRuntimeClass() const;
    static void _stdcall Construct(void* p);
```

Example 1: (a) The beginning of a class declaration; (b) after preprocessing, the *DECLARE_DYNCREATE* macro expands to several new class members.

PROGRAMMERS TRAINING ALLIANCE

1-800-862-7280

The most convenient and efficient way to locate and register for Windows® Programmer Training Classes nationwide.

The Programmers Training Alliance is an association of independent Authorized Education Centers located in cities throughout the U.S. and Canada. Every month the Alliance offers hundreds of hands-on training classes. Classes are available at beginner, intermediate, and advanced levels. Course topics include:

- Windows NT
- Supporting Windows 95
- Visual Basic
- Visual C++
- Access
- SQL Server
- MS Mail
- Lotus Notes
- Power Builder
- Novell Netware
- Borland Delphi
- Oracle Workgroup

Just one call and you're set! Our Training Consultants will assist with class selection and provide course outlines within minutes. We will locate the training center closest to you and register you in the class. Or we can send an experienced instructor to teach on-site at your office. Leave the work to us. It has never been so simple! For more information, call us.

3000 Old Alabama Road, Ste. 119-200
Alpharetta, Georgia 30022
E-Mail 74271.1420 @compuserve.com
Phone 404-772-0799
FAX 404-772-0354



CIRCLE NO. 230 ON READER SERVICE CARD

(continued from page 64)

This mechanism allows on-the-fly registration of object types whenever they are linked into the program, thus solving Problem 2.

A common question is, What is the difference between the various DECLARE and IMPLEMENT macros? All DECLARE, DYNAMIC and IMPLEMENT_DYNAMIC macros define a static instance of *CRunTimeClass* similar to the DYNCREATE shown earlier, except that the *Construct* field is NULL. DECLARE, DYNCREATE and IMPLEMENT_DECLARE add the *Construct* function for dynamic type creation. DECLARE_SERIAL and IMPLEMENT_SERIAL build on the DYNCREATE macros, and replace the 0xFFFF entry with the structure's schema number.

The SERIAL macros also define *operator>>* for the class. The *operator>>* requires special handling because it is called with a pointer to the class, but no instance of the class will exist until the instance has been serialized in from the file. Without an instance of the class, MFC cannot access the run-time class information to ensure that the object being loaded is equivalent to or is a derived class of the given pointer. By overloading *operator>>*, MFC is able to pass in a pointer to the run-time type information so that the serialization mechanism will be typesafe.

Creating Types from a File

The third problem is to create a mapping scheme to allow a type to be created based on information read from a file. Given that the compiler requires class names to be unique and that the class name is already embedded in the *CRunTimeClass* structure, the actual class name is the ideal candidate to write to a file in order to identify a class.

An object could be saved to an archive by writing the class's name and data. MFC does this and a little bit more for each class it encounters. The name of the class is from the structure's *CRunTimeClass*, which is obtained from a virtual function in the object. Because the typing is dynamic and done at run time, a structure of type *Tiger* will be properly written even if MFC is passed a pointer to a base class of type *Animal*. This type safety is important. Any function can safely save an object to an archive even if the object's exact type is unknown.

The same benefit applies to restoring an object from an archive. Going back to the example at the beginning of the article, MFC is able to successfully load the correct document type from a file with the simple statement in Example 4.

In the implementation of *operator>>*, MFC loads the name of the class from the file, then searches the list of types for that

name. As long as the type exists in the registry and was declared with either DECLARE_DYNAMIC or DECLARE_SERIAL, MFC can construct the object. The actual loading of the data appropriate to that kind of object is delegated to the object itself by calling its *Serialize* virtual member function, and Problem 3 is solved.

The type of object created is separated from the type of object requested. If a derived class is loaded into a pointer to a base class, as in the example of *CDocument*, then the correct derived class will still be created. This is the only way that the correct *Vtbl* pointer can be set up to point at the object's virtual functions. If the object in the archive is not a "kind of" the specified object, MFC will throw an exception.

There are two potential pitfalls here for developers. First, renaming a serializable structure or class will corrupt any old save files or archives. Second, MFC does not record the length of each object into the archive. If MFC can't load an object, it will not be able to skip the object and load the rest of the archive.

Optimizing Archives

When I first scanned this code, I had visions of massive bloat in the save file and painful delays while the linked list was walked repeatedly. But MFC keeps its archive size down and its execution speed up by using hashed identifiers.

MFC keeps a hash table that tracks all classes and objects written to an archive. Once they are written, MFC does not rewrite them; it writes an identifier instead. Thus, when the archive is read back in, the linked list of types is traversed only for a new class. For subsequent instances of the class, the proper instance of *CRunTimeClass* will be found with a hashed lookup.

This behavior also means that multiple references to the same object are handled correctly. If objects A and B both point to a single instance of object C when the archive is created, they will both point to a single instance of object C when the file is read back in. MFC will also correctly resolve circular references between objects.

The implementation is much faster than I expected. On a 486/66, MFC was able to save and load an archive over a megabyte long with 10,000 separate instances of *Carry<DWord,DWord>* in under two seconds.

An important limitation is that the hash table can have no more than 32,766 classes and objects per archive context. This count only includes classes derived from *CObject* and serialized with *operator<<*, not fundamental types such as *short* and *long*, *CString*, and *CPoint*. (See MFC Technical Note 2: Persistent Object Data Format for more information on how archives are constructed.)

```

CDocument* pDoc;
CArchive ar;

ar >> pDoc;

```

Example 4: Loading the correct document type from a file.

Schema Versions

A poorly documented feature in 32-bit MFC 3.2 is support for versionable schemas, where MFC allows the *Serialize()* routine to handle the various versions of the class instead of throwing an exception. This feature is very important in an evolving project. Although I will describe how to implement a versionable schema, the implementation is broken in Visual C++ 2.x and fails at run time. I recommend telling Microsoft how much you would like this feature fixed.

In MFC, each structure that uses *DECLARE_SERIAL* and *IMPLEMENT_SERIAL* has an associated version number. This number is normally set to 1, as shown in most MFC sample code; for example, *IMPLEMENT_SERIAL(CScribDoc, CObject, 1)*.

Each structure or class has its own version number, which can vary independently of the others. MFC automatically writes the version number into the archive after the class ID. Prior to 3.0, MFC did not have a mechanism to completely support this version number, so older versions throw an exception if the schema number of the object in the file does not match the current schema number. This inhibits support for multiple schemas.

In MFC 3.0 and later, this behavior is only the default, and can be changed. By OR'ing the third parameter of the *IMPLEMENT_SERIAL* macro with the constant *VERSIONABLE_SCHEMA*, MFC will allow you to handle the schema version in your *Serialize()* function. For example, to set the document version number to 3, specify *DECLARE_SERIAL(CScribDoc, CDocument, VERSIONABLE_SCHEMA(3))*.

To use this feature, a class should call *GetObjectSchema()* in its *Serialize()* member function when the archive is loaded; see Listing Three.

Conclusion

By creating a foundation of a run-time type mechanism with a class factory and building serialization on top of it, MFC implements a fast, flexible, typesafe serialization mechanism. This mechanism should be powerful enough to satisfy most design requirements.

DDJ

(Listings begin on page 122.)

YOU ALREADY USE OUR PRODUCTS EVERY DAY



SINCE 1979 THE WORLD'S LEADING COMPANIES HAVE BUILT THEIR APPLICATIONS WITH OUR TECHNOLOGY

THE PROFESSIONAL DBMS

FOR YOUR NEXT PROJECT CONSIDER FAIRCORN:
THE BEST PRICE FOR THE BEST PERFORMANCE

c-tree Plus®

ROYALTY-FREE HIGH PERFORMANCE DBMS

By simply linking with a different c-tree Plus library, an application can move from a single-user application to a multi-user application, to a client-server application. c-tree Plus is distributed in complete C source code and is known for its unparalleled flexibility, portability and royalty-free licensing policy. This licensing puts your development budget to work for you. c-tree Plus provides "makes" for the following platforms: DOS (Microsoft C, Borland C, Symantec, and for Zortech C and Watson C, 16/32-bit); Microsoft Windows (Microsoft C, Borland C; Microsoft NT; OS/2 (Microsoft C, IBM C/SET, Borland OS/2); Unix; GXX (16 and 32-bit); COHERENT; IBM RS6000; SUN; Motorola 68000; HP9000; Xenix; Apple Macintosh and DEC Alpha.

- COMPLETE "C" SOURCE CODE
- SINGLE / MULTI USER
- CLIENT / SERVER (optional)
- FULL ISAM FUNCTIONALITY
- NO ROYALTIES
- TRANSACTION PROCESSING
- FIXED / VARIABLE LENGTH RECORDS
- MULTIPLE KEYS
- DYNAMIC SPACE RECLAMATION
- HIGH SPEED DATA / INDEX CACHING
- BATCH OPERATIONS
- UNSUPPRESSED PORTABILITY

SERVERS SQL & non-SQL

Faircom Servers are available for the following platforms: DOS, Windows, Windows NT, MUM, OS/2, AT&T Unix, SCO Unix, Interactive Unix, GXX, Banyan, HP9000, SUN, Solaris, IBM RS6000, Motorola 68000, Apple A/UX, Apple System 7, DEC Alpha. Faircom Servers communicate using NETBIOS, SPX/IPX, TCP/IP, shared memory, message queues or StreetTalk depending on the server platform. Most server platforms support more than one protocol providing you the flexibility to choose how your client applications communicate to the server. For example, the Faircom DS/2 server can communicate with shared memory and NETBIOS clients at the same time, and most of the Unix packages include a shared memory and a TCP/IP Faircom Server.

- CLIENT / SERVER MODEL
- TRANSACTION PROCESSING
- HETEROGENEOUS NETWORK
- REQUIRES < 2MB RAM
- ON-LINE BACKUP
- DISASTER RECOVERY
- ROLL BACK - FORWARD
- ANTI-DEADLOCK RECOVERY
- CLIENT-SIDE "C" SOURCE
- MULTI-THREADED
- OEM (UPON REQUEST)

Server®



r-tree®

REPORT GENERATOR

The r-tree Report Generator handles virtually every aspect of report generation.

- COMPLETE "C" SOURCE CODE
- COMPLEX MULTIPLE REPORTS
- MULTI-FILE ACCESS
- COMPLETE LAYOUT CONTROL
- CONDITIONAL PAGE BREAKS
- NESTED HEADERS AND FOOTERS
- DYNAMIC FORMAT SPECIFICATIONS
- HORIZONTAL REPEATS
- AUTOMATIC ACCUMULATORS



d-tree™

APPLICATION DEVELOPMENT

d-tree provides the convenience of a 4GL product at the C development tool level.

- COMPLETE "C" SOURCE CODE
- PROTOTYPED GENERATION
- DATA DICTIONARY
- PORTABLE SCREEN HANDLER
- RUNTIME RESOURCE CONTROL
- FLEXIBLE DATA WINDOWS
- DATA FILE REFORMATTING
- EASY TO USE DATA INTERFACE
- MUCH MUCH MORE

THE UNIQUE SOLUTION: ONE PRODUCT FOR OVER 100 PLATFORMS



FAIRCORN®
corporation

CALL TODAY FOR MORE INFORMATION
(800)
234-8180

U.S.A.
EUROPE
JAPAN

4006 W.Broadway - Columbia, MO 65203 U.S.A. - phone (314) 445-6893 - fax (314) 445-6998
Via Salsomano 15/17 - 24021 Albino (BG) - ITALY - phone (035) 773-464 - fax (035) 773-806
KEDA Bldg. #34F - 112G, Komacheru - Tokyo, ME 514 Japan - phone (03) 25-7504 - fax (03) 24-9723

CIRCLE NO. 84 ON READER SERVICE CARD

(continued from page 120)

```

        return NULL;
    } else {
        return NULL;
    }
}

BOOL __export KMetaClass::CheckNextStrict( KFile* f )
{
    ASSERT( k_attached );
    return KMetaClass::loadNext( f ) == this;
}

BOOL __export KMetaClass::IsSubClass( char *name )
{
    // Relationship is 'receiver IsSubClassOf name'
    if ( name == m_name ) {
        return TRUE;
    } else if ( m_super_class != NULL ) {
        return m_super_class->IsSubClass( name );
    } else {
        TRACE( "KMetaClass::IsSubClass - %s is not a sub-class of"
            " %s.\n", m_name, name );
        return FALSE;
    }
}

KMetaClass * __export KMetaClass::Find( char *name, BOOL search_alis )
{
    KMetaClass *k = k_class_list;
    KMetaClassAlias *a = k_class_list;
    BOOL f = FALSE;
    //ASSERT( k_attached );
    while ( k != NULL && f ) {
        if ( k->m_name == name ) { strcmp( k->m_name, name ) == 0 ) {
            } else {
                k = k->m_next_class;
            }
        }
        if ( k == NULL && search_alis ) {
            // Search aliases
            while ( a != NULL && !f ) {
                if ( strcmp( name, a->m_alis_name ) == 0 ) {
                    k = Find( a->m_class_name, FALSE );
                    f = TRUE;
                } else {
                    a = a->m_next_alis;
                }
            }
        }
    }
}

typedef _DPSUG
if ( k == NULL ) {
    if ( search_alis ) {

```

```

        TRACE( "KMetaClass::Find - failed to find class:"
            " %s in class list or in alias list\n", name );
    } else {
        TRACE( "KMetaClass::Find - failed to find class:"
            " %s in class list\n", name );
    }
}

#endif
return k;
}

void __export KMetaClass::AttachSubClass( KMetaClass *k )
{
    Kc->AttachSibling( m_sub_class );
    k->m_super_class = this;
    m_sub_class = k;
}

void __export KMetaClass::AttachSibling( KMetaClass *k )
{
    m_sibling_class = k;
}

// Meta class alias code.
__export KMetaClassAlias::KMetaClassAlias( char *class_name, char *alis_name )
{
    m_alis_name = alis_name;
    m_class_name = class_name;
    m_next_alis = k_alis_list;
    k_alis_list = this;
}

__export KMetaClassAlias::KMetaClassAlias( void )
{
}

```

MFC

Listing One

```

struct CRuntimeClass
// Attributes
LPCTSTR m_lpszClassName;
int m_nObjCtorSize; // schema number of the loaded class
void (PASCAL* m_pfnConstruct)(void* p); // NULL = abstract class
CRuntimeClass* m_pBaseClass;

// Operations
Object* CreateObject();
// Implementation
BOOL ConstructObject(void* pThis);
void Store(ARCHIVE* ar);
static CRuntimeClass* PASCAL Load(
    ARCHIVE* ar, UINT* pSchemaNum);
// CRuntimeClass objects linked together in simple list
CRuntimeClass* m_pNextClass; // linked list of registered classes
};

```

Listing Two

```

void __stdcall CScriptDoc::Construct(void* p)
{
    new(p) CScriptDoc;
}

CRuntimeClass* __stdcall CScriptDoc::GetBaseClass()
{
    return (&CDocument::classCScriptDoc);
}

CRuntimeClass CScriptDoc::classCScriptDoc = {
    "CScriptDoc",
    sizeof(CScriptDoc),
    CScriptDoc::Construct,
    CScriptDoc::GetBaseClass, 0 };
static const AFX_CLASSINFO* classCScriptDoc(CScriptDoc::classCScriptDoc);
CRuntimeClass CScriptDoc::GetRuntimeClass() const
{
    return &CScriptDoc::classCScriptDoc;
}

```

Listing Three

```

class CSerialObject : public Object {
    DECLARE_SERIAL(CSerialObject);
    DYNAMIC_SERIAL; // used to be unsigned short in version 1
};

IMPLEMENT_SERIAL(CSerialObject, Object, VERSIONABLE_SCHEMA, 2);
CSerialObject::Serialize(ARCHIVE* ar)
{
    if (ar.IsStoring()) {
        ...
    } else {
        WORD nVersion = ar.GetObjCtorSchema();
        switch (nVersion) {
            case -1:
                // 1 shows that the structure was created with DYNAMIC, not
                // with SERIAL. Setting this value is probably an error.
                break;
            case 1: // This version used unsigned short
                unsigned short oldval;
                ar >> oldval;
                nValue = oldval;
                break;

```

GRAF/DRIVE PLUS 4.5 Printer/Plotter Graphics for DOS

Publication-quality hard copy graphics for Microsoft-compatible C compilers, Borland C, and Borland Pascal. Borland programmers use the same functions for screen and hard copy graphics, with almost no change to their source code. Also includes Borland SVGA drivers with mouse support. You'll see our drivers in commercial software for desktop publishing, business charts, investment analysis, maps, and science and engineering. *"Highly recommended"*—Jeff Duntzman, Dr. Dobbs.

Full-resolution output to LaserJet, Epson & IBM dot matrix, Deskjet, DeskJet Color, PankJet, PostScript (mono/color), Canon laser & Ink Jet, Epson Stylus, HP plotters, color dot matrix, obscure printers too numerous to mention, PCX, TIF, DXF, WMF, BMP, AI, CGM, more file formats. (DTP add-on req'd for file formats and less common printers.)

More than 140 functions, including arc, bar, bar3d, circle, drawpoly, fillpoly, ellipse, fillfills,

line, outtext, pie slice, pulpcx, putimage, rectangle, sector, setcolor, setfillpattern, setfillstyle, setlinestyle, setpalette, settextjustify, setviewport, textheight, textwidth.

Print to LPT1-4, COM1-4 or disk, portrait or landscape, any size from 1 inch to 5 feet • Use PostScript fonts on a PostScript printer • Use resident or downloaded fonts on a LaserJet • Use default and scalable stroke fonts on any printer • Print spooler • Print both vector and raster graphics • Scale, dither, and print PCX, TIFF, BMP, large, GIF files, and screen dumps • Print without spawning a separate program (except Borland DPMI-32) • Print without clearing your graphics or text screen • Network-compatible • Does not require a graphics screen • Includes files for real mode, 16-bit protected mode, and Borland DPMI-32.

Personal License \$149, Developers with royalty-free distribution \$299. DTP Drivers \$99. 30-day m/b guarantee.

FLEMING SOFTWARE BOX 569 OAKTON, VA 22124
(703) 591-6451

CIRCLE NO. 85 ON READER SERVICE CARD

```

case 2;
// Current version uses DWORD
of >> m.value;
break;
default:
// Bogus value - probably corrupt data file.
break;
}
}

```

ENVIRONMENT VARIABLES

Listing One

```

/* envvars.c: This prints out the current env[] set and the current Windows
default environment settings. I used this to determine if a change to the
environment settings had taken place. Build this as a Borland EasyWin
(Windows command line) executable. 5/95 Fritz Lowrey */

#include <stdio.h>
int main(int argc, char *argv[], char *envp[]) {
    int i;
    char *denv; /* default environment */

    printf("Program environment array:\n");
    for (i=0; envp[i] != NULL; i++)
        printf("%s\n", envp[i]);
    printf("Windows default environment:\n");
    denv = GetDefaultEnvironment();
    while (*denv != NULL) {
        printf("%s\n", denv);
        denv += strlen(denv) + 1; /* move to the next string */
    }
    exit(0);
}

```

Listing Two

```

/* winenv.c: Build using Borland C++ EasyWin environment to allow for stdio
function calls. Copyright John "Fritz" Lowrey, 24 May, 1995. This code and
research that it is possible were done in conjunction with the University of
Southern California University Computer Services Dept. */

#include <windows.h>
#include <stdio.h>
#include <string.h>
#include <dos.h>

#define DIRM

/* static variable for environment manipulation, not visible to other modules */
static char *lpNewEnv; /* pointer to environment space */
static HGLOBAL hNewEnv; /* handle to environment memory */
static int ENVSIZE; /* size of environment space */

/* LDRPAPR structure needed by LoadModule */
struct LDRPAPR {
    WORD segEnv; /* child environment */
    LPSTR lpOrigLine; /* child command tail */
    UINT FAR lpShow; /* how to show child */
    UINT FAR lpReserved; /* must be NULL */
};

/* Initialize the environment space, ENVSIZE is the size of the environment
region (defined on the SHELL line of CONFIG.SYS) */
/* returns -1 on error or 0 on success */
int EnvInit(int esize) {
    ENVSIZE = esize;
    if ((hNewEnv = GlobalAlloc(GPTR, GROW_SHARE, ENVSIZE)) == NULL)
        return -1;
    if (lpNewEnv = GlobalLock(hNewEnv)) == NULL)
        return -1;
    /* we now have a pointer to the memory, fill it from the env space */
    if (memcpy(lpNewEnv, GetDefaultEnvironment(), ENVSIZE) == NULL)
        return -1;
    /* environment space is initialized, return 0 */
    return 0;
}

/* definitions for new getenv and putenv routines */

/* Simple new getenv() routine. Search must be a label only */
LPSTR NewGetEnv(LPSTR search) {
    LPSTR tmpstr;

    /* point tmpstr at the environment space */
    tmpstr = lpNewEnv;
    /* scan through the space */
    while (tmpstr[0] != NULL) {
        /* if "search" is found at beginning of tmpstr, return tmpstr */
        if (strchr(tmpstr, search) == tmpstr)
            return tmpstr;
        tmpstr += strlen(tmpstr) + 1; /* move to next string */
    }
    /* if we fall through to here, return NULL */
    return NULL;
}

/* new putenv() routine. returns 0 on success -1 on failure */
int NewPutEnv(LPSTR putstr) {
    LPSTR currentloc; /* current location in the buffer */
    LPSTR tmpstr;
    char label[80]; /* the label portion of putstr */

    (continued on page 124)

```

Windows 95 without the risk!

System Commander® makes it safe and easy to add as many operating systems to your PC as you want!

Quick and Easy Installation

System Commander will have your PC ready to add new operating systems in less than 5 minutes. It does not require repartitioning, nor a partition of its own.

The first reboot brings up a menu of the operating systems already installed. Select the one you want and System Commander does the rest.

Saves You Time and Effort

As you install new operating systems, System Commander saves the key files and adds the new OS to the System Commander menu.



System Commander makes it safe and easy to evaluate new operating systems without giving up the reliability of your existing OS.

Saves You Money!

Instead of investing thousands of dollars in new PCs, you can now have as many as 100 operating systems on one PC with up to 14 hard drives of any size or type.

System Commander is only \$99.95 and comes with a 60 day money-back guarantee. For a limited time, get FREE overnight shipping when you mention this ad.

System Commander

Call now!
1-800-648-8266
60-DAY MONEY-BACK GUARANTEE



V Communications, Inc.
4320 Stevens Creek Blvd., Suite 120-DD
San Jose, CA 95129 408-296-4424
FAX 408-296-4441

*When ordered before noon PST. Extra charge for Saturday delivery. Standard shipping outside US. CA residents add \$7.25 sales tax. Offer subject to change without notice. All logos and product names are trademarks of their respective companies. VISAMC/Amex/COD © 1995

(continued on page 124)